Microarchitectural Mechanisms to Exploit Value Structure in SIMT Architectures

Ji Kim, Christopher Torng, Shreesha Srinath, Derek Lockhart, and Christopher Batten

School of Electrical and Computer Engineering, Cornell University, Ithaca, NY
{jyk46,clt67,ss2783,dml257,cbatten}@cornell.edu

ABSTRACT

SIMT architectures improve performance and efficiency by exploiting control and memory-access structure across data-parallel threads. Value structure occurs when multiple threads operate on values that can be compactly encoded, e.g., by using a simple function of the thread index. We characterize the availability of control, memory-access, and value structure in typical kernels and observe ample amounts of value structure that is largely ignored by current SIMT architectures. We propose three microarchitectural mechanisms to exploit value structure based on compact affine execution of arithmetic, branch, and memory instructions. We explore these mechanisms within the context of traditional SIMT microarchitectures (GP-SIMT), found in general-purpose graphics processing units, as well as fine-grain SIMT microarchitectures (FG-SIMT), a SIMT variant appropriate for compute-focused data-parallel accelerators. Cycle-level modeling of a modern GP-SIMT system and a VLSI implementation of an eight-lane FG-SIMT execution engine are used to evaluate a range of application kernels. When compared to a baseline without compact affine execution, our approach can improve GP-SIMT cycle-level performance by 4-17% and can improve FG-SIMT absolute performance by 20–65% and energy efficiency up to 30% for a majority of the kernels.

Categories and Subject Descriptors

C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures—*parallel processors, SIMD*; I.3.1 [**Computer Graphics**]: Hardware Architecture—*graphics processors*

General Terms

Design, Performance

1. INTRODUCTION

General-purpose graphics-processing units (GPGPUs) are growing in popularity across the computing spectrum [2, 28, 35]. Most GPGPUs use a single-program multiple-data (SPMD) model of computation where a large number of independent threads all execute the same application kernel [25, 27, 29]. Although these SPMD programs can be mapped to the scalar portion of general-purpose multicore processors (an active area of research [11, 32]), architects can improve performance and efficiency by using specialized data-parallel execution engines to exploit the structure inherent in SPMD programs. Examples include compiling SPMD programs to subword-SIMD units [5, 18, 19], to more traditional vector-SIMD units [2, 17], or to single-instruction multiple-thread (SIMT) units [1, 16, 22, 28].

We focus on SPMD programs mapped to SIMT microarchitectures which use an explicit representation of the SPMD model: a SIMT kernel of scalar instructions is launched onto many *data-parallel threads*. Threads use their thread index to work on disjoint data or to enable different execution paths. Threads are mapped to an architecturally transparent number of hardware thread contexts to enable scalable execution of many kernel instances in parallel. A SIMT microarchitecture usually includes several *SIMT engines*; each engine is responsible for managing a subset of the threads, including its own inner levels of the memory hierarchy, and is relatively decoupled from the other engines.

SIMT microarchitectures exploit *control structure* and *memory-access structure* in SPMD programs to improve performance and area/energy efficiency. Control structure characterizes how often multiple threads execute the same instruction in the kernel, while memory-access structure characterizes the regularity of inter-thread addresses for the same load/store instruction. To exploit control structure, a SIMT engine executes a set of consecutively indexed threads (usually called a *warp* or *wavefront*) in lock-step on the SIMT engine, amortizing control overheads (e.g., instruction fetch, decode, interlocking) and hiding execution latencies. Performance and efficiency are maximized when all threads take the same path through the kernel. To exploit memory-access structure, a memory coalescing unit dynamically compares memory requests made by threads and merges multiple scalar memory requests into one wide memory request, amortizing control overheads in the memory system (e.g., arbitration, tag check, miss management) and reducing bank conflicts. Performance and efficiency are maximized when all threads access the same or consecutive addresses.

In this work, we study GPGPU SIMT architectures (GP-SIMT) and a fine-grain SIMT variant more appropriate for compute-focused data-parallel accelerators (FG-SIMT). Our GP-SIMT microarchitecture is modeled after an NVIDIA Fermi-class graphics processor [28]. Our FG-SIMT microarchitecture is a new approach targeted towards general-purpose data-parallel accelerators focused purely on computing as opposed to graphics rendering. FG-SIMT is designed for improved performance and energy efficiency on more irregular kernels that require fewer data-parallel threads. FG-SIMT might be a viable alternative for use in future Intel Many Integrated Core (MIC) accelerators [17]. Currently these accelerators include tens of general-purpose processors each with a wide vector-SIMD coprocessor. FG-SIMT can enable a simple programming model (e.g., handling complex control flow or modular function calls from within data-parallel threads) yet still achieve much of the benefit of more traditional vector-SIMD execution.

Prior work by Collange et al. introduced the concept of *value structure* in SIMT kernels and used static and dynamic analysis to

ISCA'13, June 23–27, 2013, Tel Aviv, Israel.

characterize value structure in CUDA programs [5, 7, 8]. Value structure refers to situations where threads execute the same instruction operating on data that can be described in a simple, compact form. For example, all threads might operate on the same value or on values that can be represented with a function of the thread index. In vector-SIMD microarchitectures (e.g., Intel MIC [17], AMD Graphics Core Next [2]), value structure is exploited by refactoring work onto a scalar control processor, but the SIMT thread-focused model leads to unexploited value structure when threads calculate common expressions, execute inner loops, or generate structured addresses. In this paper, we extend prior studies on value structure in GP-SIMT kernels with our own detailed study of value structure in FG-SIMT kernels. These studies motivate our interest in exploring new microarchitectural mechanisms to exploit value structure in SIMT architectures.

Previous dynamic hardware schemes in this area include: tracking affine values derived from shared loads or thread index and reusing the SIMT register file and lane functional units to execute affine arithmetic (but without support for expanding compact values after divergence) [7]; tracking identical values with many value comparisons across threads in a warp (but without support for affine values) [15]; and tracking affine values in load/store data for compact storage in the memory system [8].

In this paper we propose a detailed implementation of the affine value tracking technique sketched in [7], but in addition we describe a new way to exploit value structure using compact affine execution on a separate affine functional unit which supports lazily expanding compact values after divergence. We also propose support for affine conditional branches, which avoid lengthy branch resolution, and affine memory operations, which allow us to infer vector memory operations in a SIMT architecture. We use cycle-level modeling within the GPGPU-Sim framework [3] to estimate the performance impact of our technique in GP-SIMT architectures. Since most GP-SIMT microarchitecture and layout details are not public, we focus our detailed evaluation of cycle time, area, and energy on FG-SIMT architectures. We have developed a register-transfer-level (RTL) model of an eight-lane FG-SIMT engine with a corresponding L1 memory system, and we use commercial ASIC CAD tools to synthesize and place-and-route the design in a modern TSMC 40 nm process. Using these models, we present a variety of results illustrating the benefits and some of the trade-offs involved in exploiting value structure in SIMT architectures.

The contributions of this paper are as follows. We describe three novel mechanisms to efficiently execute: (1) affine arithmetic with support for lazy expansion of compact values after divergence; (2) affine conditional branches with support for both uniform and affine operands; and (3) affine memory operations which avoid the need for memory coalescing to generate full-warp vector memory accesses. We demonstrate these mechanisms in both GP-SIMT and FG-SIMT architectures, and evaluate the cycle time, area, and energy impact of these techniques on a VLSI implementation.

2. STRUCTURE IN SIMT KERNELS

In this section, we illustrate and quantify control, memory-access, and value structure in SIMT kernels using a simple example and statistics from high-level functional simulation.

2.1 Example Application Kernel

Figure 1 shows C code and compiled assembly for a simple SIMT kernel. Figure 1(a) shows a CUDA implementation, and Figures 1(c)–(d) show the corresponding PTX assembly and machine assembly for an NVIDIA Fermi-class GPGPU. PTX is usually just-in-time compiled to the underlying machine instruction set. Figure 1(b) shows the same example implemented using our lightweight C++ FG-SIMT programming framework. The initialization function manages kernel parameters and returns the thread index. In contrast to GP-SIMT architectures, FG-SIMT architectures execute the kernel launch code on a control processor tightly integrated into the SIMT engine. A software run-time can manage work distribution across multiple SIMT engines. Figure 1(e) shows the corresponding assembly which uses a simple RISC instruction set for both the control processor and data-parallel threads.

2.2 Control and Memory-Access Structure

Lines 2–3 in Figures 1(a)–(b) have significant control structure; all threads execute in lockstep. The branch associated with line 4 may or may not exhibit control structure depending on the data in the `y` array. Note that in the FG-SIMT assembly, this control flow corresponds to a scalar branch instruction while in the GP-SIMT machine assembly, it corresponds to predicated instructions.

Line 3 in Figures 1(a)–(b) also has significant memory-access structure; all threads load data from consecutive elements. The store on line 5 has less memory-access structure; it translates into a conditional store or a store under branch, and thus threads may or may not store the max value to consecutive elements.

2.3 Value Structure

Value structure occurs when values used by the same operation across threads can be represented in a simple, compact form. For example, on line 7 of Figure 1(c) the `mad` instruction operates on three registers with values that are a simple function of the thread index. Similarly, lines 2–4 in Figure 1(e) load the thread index into a register, shift it by a constant, and then add the result to the array base pointer. These instructions operate on *affine values* that can be compactly represented in the following form:

$$V(i) = b + i \times s$$

where i is the thread index, b is the base, and s is the stride. Affine values with a stride of zero are a special case in which all the numbers in the sequence are identical, and we term these *uniform values*. We can exploit this kind of value structure by compactly encoding affine values as a base/stride pair.

Certain arithmetic operations can be performed directly on this compact affine representation to produce an affine result. We term this *affine arithmetic*. For example, an affine addition can be computed as follows:

$$V_0(i) = b_0 + i \times s_0 \qquad V_1(i) = b_1 + i \times s_1$$
$$V_0(i) + V_1(i) = (b_0 + b_1) + i \times (s_0 + s_1)$$

As another example, affine multiplication is possible if at least one of the operands is uniform:

$$V_0(i) = b_0 + i \times s_0 \qquad V_1(i) = b_1$$
$$V_0(i) \times V_1(i) = (b_0 \times b_1) + i \times (s_0 \times b_1)$$

It is straight forward to develop a system of rules for affine addition, subtraction, shifts, and multiplication (note that [7] does not consider affine multiplication). Using these rules, we have labeled the instructions in Figure 1 that are able to operate directly on compactly encoded affine values. A significant fraction of the instructions are affine across all assembly sequences. This is a fundamental consequence of SIMT-based microarchitectures; threads likely work on structured values to calculate the location of their input/output values (e.g., lines 2–4 in Figure 1(e)) and to compute

```
1 __global__ void ex_gpsimt_kernel( int y[], int a ) {
2   int idx = blockIdx.x*blockDim.x + threadIdx.x;
3   y[idx] = a * y[idx];
4   if ( y[idx] > THRESHOLD )
5     y[idx] = Y_MAX_VALUE;
6 }
```

(a) GP-SIMT CUDA Kernel

```
1 __fgsimt__ void ex_fgsimt_kernel( int y[], int a ) {
2   int idx = fgsimt::init_kernel( y, a );
3   y[idx] = a * y[idx];
4   if ( y[idx] > THRESHOLD )
5     y[idx] = Y_MAX_VALUE;
6 }
7
8 void ex_fgsimt( int y[], int a, int n ) {
9   fgsimt::launch_kernel( n, &ex_fgsimt_kernel, y, a );
10 }
```

(b) FG-SIMT Kernel

```
1  u  ld.param.u32  %r2, [_ex_gpsimt_kernel_y]
2  u  ld.param.u32  %r3, [_ex_gpsimt_kernel_a]
3  u  cvta.to.global.u32 %r4, %r2
4  u  mov.u32       %r5, %ntid.x
5  a  mov.u32       %r6, %ctaid.x
6  a  mov.u32       %r7, %tid.x
7  a  mad.lo.s32    %r8, %r5, %r6, %r7
8  a  shl.b32       %r9, %r8, 2
9  a  add.s32       %r1, %r4, %r9
10 a  ld.global.u32 %r10, [%r1]
11 g  mul.lo.s32    %r11, %r10, %r3
12 a  st.global.u32 [%r1], %r11
13 g  setp.gt.s32   %p1, %r11, THRESHOLD
14 u  @%p1 bra      L1
15 u  ret
16 u L1: mov.u32    %r12, Y_MAX_VALUE
17 a  st.global.u32 [%r1], %r12
18 u  ret
```

(c) PTX "Virtual" Assembly

```
1  u  mov          R1, c[0x1][0x100]
2  a  s2r          R0, SR_CTAid_X
3  a  s2r          R2, SR_Tid_X
4  a  imad         R0, R0, c[0x0][0x8], R2
5  a  iscadd       R3, R0, c[0x0][0x20], 0x2
6  a  ld           R0, [R3]
7  g  imul         R0, R0, c[0x0][0x24]
8  g  isetp.gt.and P0, pt, R0, THRESHOLD, pt
9  a  st           [R3], R0
10 u  @p0 mov32i   R2, Y_MAX_VALUE
11 a  @p0 st       [R3], R2
12 u  exit
```

(d) Fermi Machine Assembly

```
1  u  lw    y_ptr, y_base_addr
2  a  tidx  i
3  a  sll   i, i, 0x2
4  a  addu  y_ptr, y_ptr, i
5  a  lw    y, 0(y_ptr)
6  u  lw    a, a_addr
7  g  mul   y, y, a
8  g  slti  t, y, THRESHOLD
9  a  sw    y, 0(y_ptr)
10 g  bnez  t, done
11 u  li    y, Y_MAX_VALUE
12 a  sw    y, 0(y_ptr)
13   done:
14 u  exit
```

(e) FG-SIMT Assembly

Figure 1: Example SIMT Code and Compiled Assembly – Kernel multiplies input array by constant and then saturates to threshold. Explicit bounds check elminated for simplicity. Assembly instructions are manually labeled according to the ability for that dynamic instruction to ideally exploit value structure: `g`: generic, cannot exploit value structure; `u`: uniform, affine stride is zero; `a`: affine with non-zero stride.

similar intermediate values (e.g., line 11 in Figure 1(e)). In addition to affine arithmetic, value structure can also have a direct impact on control structure and memory-access structure. For example, if a kernel has an inner loop where the number of iterations is the same across all threads, then the corresponding backwards branch exhibits uniform value structure. Furthermore, the load on line 5 in Figure 1(e) exhibits affine value structure in its source address.

Note that even though there is ample value structure in this example, not all of this structure is trivial to exploit; value structure under predication or after divergence is particularly difficult to exploit. If not all threads handle the data-dependent threshold comparison in the same way, then it is not clear that we can exploit the value structure when executing lines 11–12 in Figure 1(e).

2.4 Quantifying Structure in FG-SIMT Kernels

To quantitatively characterize structure in FG-SIMT kernels, we studied several benchmarks written using our C++ FG-SIMT programming framework and executed on a high-level functional simulator. This study is similar in spirit to previous studies on GP-SIMT applications [5, 7, 8, 21]. Preliminary hardware techniques have shown that affine values can be detected in up to 22% of dynamic instructions [7]. Compiler techniques have shown affine values can be exploited in up to 29% of dynamic instructions, with even greater potential assisted by more sophisticated hardware techniques [21].

We make two assumptions that generalize the results across many possible microarchitectures and differentiate our exploration from previous studies: (1) our simulator uses an infinitely long warp size; and (2) we post-process the results to calculate an idealistic reconvergence scheme where all threads with the same dynamic instruction identifier are assumed to execute together. In addition, we more explicitly characterize value structure after divergence, value structure in the memory address stream, and the percentage of instructions that can ideally use compact affine execution. Figure 2 illustrates the results for four benchmarks selected to reflect a range of structure types. More detail on these and the rest of our benchmarks can be found in Section 5.3.

Control Structure – For each benchmark, we illustrate control structure with a histogram of active threads. It is clear that the benchmarks in the first column (i.e., *sgemm*, *viterbi*) have significant control structure (i.e., no divergence) while the benchmarks in the second column (i.e., *bfs*, *mfilt*) have much less.

Memory-Access Structure – The load data address (LDA) and store data address (SDA) bars illustrate memory-access structure by showing the percent of loads and stores that are shared (i.e., uniform), strided (i.e., affine), or indexed (i.e., generic). *sgemm*, *viterbi*, and *mfilt* all have significant memory-access structure, while *bfs* have little memory-access structure.

Value Structure – There are a variety of different ways to characterize the value structure in these benchmarks. The load data (LDD) and store data (STD) bars indicate that some benchmarks manipulate well-structured values residing in memory (e.g., *viterbi* and *mfilt*). The register read (RRD) and register write (RWR) bars show the percentage of register accesses that can be classified as having value-structure; the benchmarks at the top of the figure have less register value structure than those at the bottom. The instruction (INST) bar clearly shows that a significant percentage of instructions in several of these benchmarks can potentially use compact affine execution.

These results illustrate how various structure types can be combined in diverse ways: *sgemm* has significant control and memory-access structure, but limited value structure. *viterbi* has significant amounts of all three kinds of structure, while *bfs* has very little structure at all. *mfilt* has less control structure, but still ample amounts of memory-access and value structure.

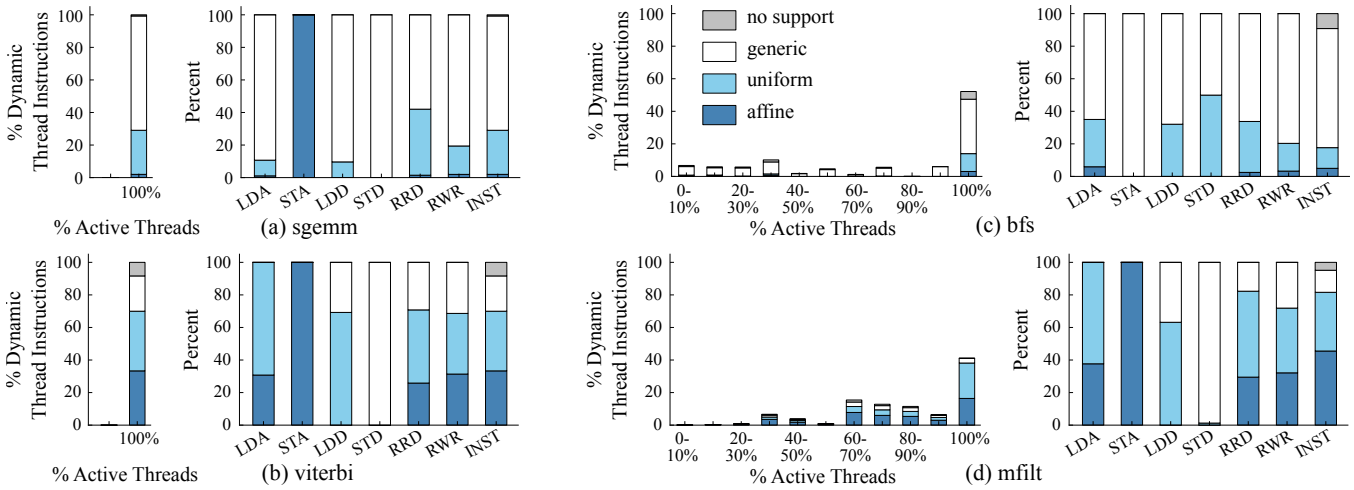

Figure 2: Characterization of Control, Memory-Access, and Value Structure in FG-SIMT Application Kernels – Benchmarks executed on high-level functional simulator assuming an infinite warp size and idealized reconvergence scheme. Histograms (left) show the distribution of active threads across all dynamic instructions. Bar plots (right) show breakdown of structure present in various aspects of the kernel including: memory-access structure in load/store addresses (LDA/STA), value-structure in load/store data (LDD/STD), value structure in register read/writes (RRD/RWR), and perecentage of instructions that can ideally use compact affine execution.

3. BASELINE MICROARCHITECTURES

In this section, we describe our baseline GP-SIMT and FG-SIMT microarchitectures.

3.1 Baseline GP-SIMT Microarchitecture

Our baseline GP-SIMT microarchitecture is based on an NVIDIA Fermi-class GPGPU with four GP-SIMT engines (called streaming multiprocessors), 16-bank configurable local memory per GP-SIMT engine (configured as 16 KB L1 data cache and 48 KB shared memory), four 786 KB L2 cache banks, fast on-chip crossbar connecting GP-SIMT engines to L2 banks, and two DRAM memory controllers [28]. Each GP-SIMT engine supports extreme temporal multithreading of up to 48 warps (32 threads/warp) and four SIMT functional units: two SIMT arithmetic units (capable of both short- and long-latency integer and floating point operations), one special functional unit, and one load/store unit. Both a 16- and 8-lane configuration are used. A large register file is implemented using many banks of single-ported SRAMs; multi-cycle read operand collectors and writeback queueing is used to manage register file bank conflicts. An aggressive memory coalescing unit aggregates cross-warp requests. The GP-SIMT issue unit can issue up to two instructions per cycle from independent warps. Two configurations for the front-end clock domain are used. One assumes the fetch, decode, and issue stages of the SIMT engine are underclocked relative to the SIMT functional units (slow front-end). The other assumes the same clock for all stages (fast front-end). Warps are fetched and issued using a fixed-priority scheduling policy. A stack-based immediate post-dominator reconvergence scheme is used [13].

3.2 Baseline FG-SIMT Microarchitecture

FG-SIMT is a SIMT variant suitable for compute-focused data-parallel accelerators (e.g., [17]) with an emphasis on efficiently executing more irregular kernels that require fewer total threads. Compared to GP-SIMT architectures, each FG-SIMT engine only executes a single warp at a time, uses highly ported register files, includes a software programmable control processor, and lacks specialized hardware for graphics rendering. FG-SIMT still includes the ability to exploit control and memory-access structure to efficiently execute warps of data-parallel threads. The primary motivation for FG-SIMT was to design an area-efficient SIMT microarchitecture that would exploit intra-warp instruction-level and data-level parallelism to hide various latencies instead of relying on extreme, inter-warp temporal multithreading. Banked, shared L1 data caches are used to increase address bandwidth for scatters and gathers. Multi-ported register files help better exploit ILP.

Figure 3 shows the microarchitecture for a FG-SIMT engine with L1 memory system. As in GP-SIMT, a large-scale FG-SIMT processor includes multiple engines interconnected through an on-chip network and outer-level memory system. While inter-engine design considerations are important, we limit our scope to a detailed implementation of a single engine to complement our system-level GP-SIMT modeling. The FG-SIMT execution engine includes the control processor (CP), FG-SIMT issue unit (SIU), eight SIMT lanes, and the FG-SIMT memory unit (SMU).

FG-SIMT Kernel Launch – The CP is a simple RISC processor with its own program counter (PC), register file, and integer arithmetic unit. The CP uses the following instruction to launch a FG-SIMT kernel:

```
launch_kernel r_n, kernel_addr
```

where `r_n` is a CP register containing the total number of threads to execute. The instruction saves the kernel start address and the return address in microarchitectural registers and initializes a warp counter to n/m where n is the value in `r_n` and m is the number of threads per warp. The instruction then initializes the active warp fragment register (AWFR) with a new *warp fragment*. A warp fragment is simply a PC and an active thread mask indicating which threads are currently executing that PC (i.e., initially the AWFR will contain the kernel address and a mask of all ones if $n >= m$).

FG-SIMT Control Processor – After launching a kernel, the CP fetches scalar instructions at the PC in the AWFR and sends them to the SIU, acting as the FG-SIMT engine's fetch unit. Unconditional jumps can be handled immediately in the CP. Conditional branch instructions are sent to the SIU, but the CP must wait for the branch resolution to determine the new active thread mask. If all threads take or do not take the branch, then the CP proceeds along the appropriate control flow path. If the threads diverge, then the

CP creates a new warp fragment with the branch target and a mask indicating which threads have taken the branch. The new warp fragment is pushed onto the pending warp fragment buffer (PWFB) before continuing execution along the not-taken path. We use a two-stack PC-ordered scheme which separates taken forward branches from taken backward branches in the PWFB to more effectively handle complicated control flow caused by intra-kernel loops [20]. The first warp brings kernel parameters into a shared load cache, reducing access latency for later warps. New warps are not scheduled until all fragments for the current warp execute to completion. Once all warps have been processed, the CP continues executing the instructions after the `launch_kernel` instruction.

FG-SIMT Issue Unit – The SIU is responsible for issuing instructions in-order to the SIMT lanes. Sophisticated scoreboarding is implemented to track various structural and data hazards and enable aggressive forwarding of data values between functional units. Load instructions are split into two micro-ops: address generation and SRF writeback. The former is issued to the lanes to generate the memory request, whereas the latter is enqueued into a SIMT load writeback queue (SLWQ). To facilitate aggressive forwarding from load instructions, writebacks are only issued to the SLU when all load responses have been received. Even though the SIU is single-issue, it is still possible (and very common) to keep multiple functional units busy, since each warp occupies a functional unit for four cycles (warp size of 32, eight lanes, four threads per lane).

FG-SIMT Lanes – Each FG-SIMT lane is composed of five SIMT functional units: two arithmetic units (SAU), an address generation unit (SGU), a store unit (SSU), and a load unit (SLU). The lane control unit manages sequencing the functional units through the threads within a warp. Both SAUs can execute simple integer operations including branch resolution; long latency operations are fully pipelined and distributed across the two units and include: integer mult, div; and IEEE single-precision floating-point add, convert, mult, div. Addresses and store data of memory requests are generated by the SGU and SSU, respectively. The SLU manages writeback of load data and the SLWQ in the SIU enables overlapped memory requests. Each FG-SIMT lane also includes a bank of the SIMT register file (SRF). The SRF is a large, multi-ported register file that supports a maximum of 32 threads per warp.

FG-SIMT Memory Unit – The SMU includes several queues for storing memory requests and responses as well as logic for memory coalescing. Each lane has a memory request queue (SMRQ) and a load data queue (SLDQ). The SMU has eight 32-bit ports to the memory system, one for each lane, in addition to a single 256-bit port used for coalesced requests and responses. Requests to the same address or consecutive addresses across lanes can be coalesced into a single wide access. Coalescing is only possible when the requests are cache line aligned. The SIMT memory response reorder queue (SMRRQ) is used to coordinate the write-back of scalar and coalesced responses and ensure that data is written back in-order. A counter is used to keep track of all in-flight memory requests, which is used to handle an implicit memory fence between the CP and SMU at the end of each kernel.

FG-SIMT L1 Memory System – The L1 memory system includes a single 16 KB instruction cache along with an eight-bank 128 KB data cache. Both caches are blocking, direct mapped, and use 32 B cache lines. Queued memory request/response crossbars use round-robin arbitration and handle per-port response reordering internally. The data cache request crossbar supports nine narrow request ports with a data size of 32 b and one wide request port with a data size of 256 b. Each data cache bank is capable of servicing one 32 b or one 256 b read/write per cycle.

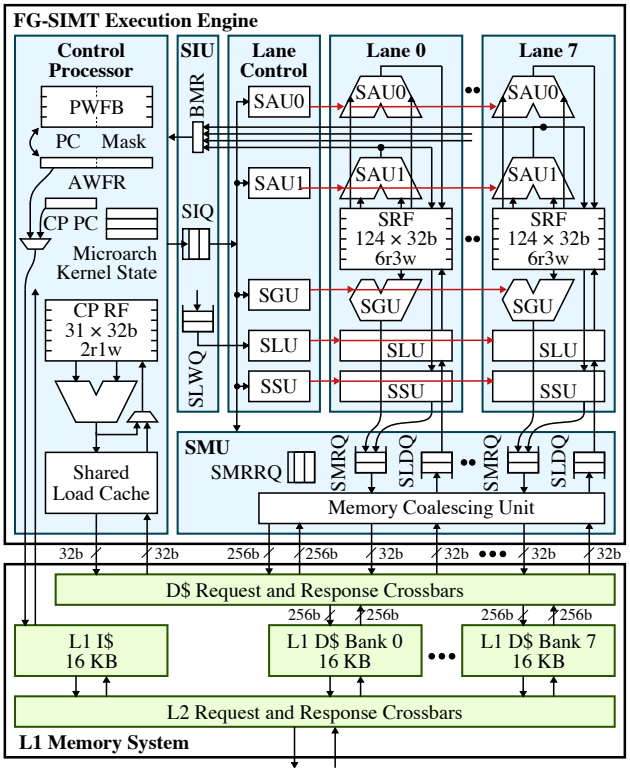

Figure 3: FG-SIMT Baseline Microarchitecture – Eight-lane FG-SIMT execution engine with L1 instruction cache and L1 eight-bank data cache. Only one branch resolution path shown for simplicity. PWFB = pending warp fragment buffer; AWFR = active warp fragment reg; CP = control proc; µArch Kernel State = kernel address reg, return address, warp counter reg; RF = regfile; SIQ = SIMT issue queue; SLWQ = SIMT load-write-back queue; SRF = SIMT regfile; SAU0/1 = SIMT arithmetic units; SGU = SIMT addr gen unit; SLU = SIMT load unit; SSU = SIMT store unit; SMRQ/SLDQ/SMRRQ = SIMT mem-req/load-data/resp-reorder queues; BMR = branch resolution mask register

4. MECHANISMS FOR EXPLOITING VALUE STRUCTURE

This section describes three mechanisms to exploit value structure: affine arithmetic, branches, and memory operations.

4.1 Detecting and Tracking Value Structure

A per-warp affine SIMT register file (ASRF) is added to the front-end of a SIMT microarchitecture. In GP-SIMT, the ASRF is inserted into the decode stage of each stream multiprocessor (SM). In FG-SIMT, the front-end is effectively the CP, and Figure 4 illustrates the required modifications. The ASRF compactly stores affine values as a 32-bit base and 16-bit stride pair (smaller stride storage is also possible to reduce area overhead especially in GP-SIMT). The ASRF has one entry for each architectural register along with a tag indicating whether that register is uniform, affine, or generic. Explicit uniform tags enables more efficiently handling affine arithmetic that is only valid for such values. The ASRF can store both integer and floating-point values, but the latter is limited to uniform values as expanding a floating-point affine value would result in an unacceptable loss of precision.

There are three ways in which a register can be tagged as uniform or affine: (1) destination of shared loads (e.g., GP-SIMT: `ld.param`

instructions; FG-SIMT loads with gp as base address); (2) destination of instructions reading the thread index (e.g., GP-SIMT: move from ntid.x; FG-SIMT: tidx instruction); or (3) destination of affine arithmetic.

4.2 Compact Affine Execution Without Divergence

We first describe a basic scheme for affine arithmetic, branches, and memory operations that cannot handle divergence. Note that this scheme can still exploit value structure after branches as long as the threads within the warp do not diverge (i.e., all threads either take or do not take the branch).

Affine Arithmetic is valid for select instructions with two uniform/affine operands as described in Section 2. Eligible instructions include addition, subtraction, multiplication, and shifts. Shift operations correspond to affine multiplication or division depending on the direction of the shift. Immediate operands are treated as uniform values. We add an affine functional unit to the front-end of a SIMT microarchitecture which is specifically designed to operate on base/stride pairs. In FG-SIMT, we can reuse the CP's standard functional unit for base computations, but still must add an extra functional unit for stride computations. Since GP-SIMT supports two issue slots, we add two sets of affine functional units for improved throughput of affine arithmetic. Each set is composed of an integer and floating point arithmetic unit, but instructions can only be issued to one of these units every cycle. The floating point unit only supports computation on uniform operands. All arithmetic units are pipelined and fully-bypassed. Eligible instructions are issued to these affine pipelines instead of the SIMT lanes. The ASRF still only needs two-read/one-write ports since we can partition the ASRF by warp; odd warps use one set of ASRF and affine functional units and even warps use the other set. Affine arithmetic executes by reading base/stride operands from the ASRF, running the computation on the affine functional units, and (potentially) writing the result back into the ASRF. If the destination register of a non-affine instruction (i.e., an instruction sent to the SIU) is marked as affine in the ASRF, that register's tag must be set to generic since the most recent value of the register will now be in the SRF. If any of the operands of a non-affine instruction are in the ASRF then the base/stride pair will be sent to the SIU along with the non-affine instruction. The SIMT lanes will use the iteratively expanded affine values instead of reading from the SRF.

Affine arithmetic is implemented in both our GP-SIMT and FG-SIMT microarchitectures. Affine arithmetic can improve both performance and energy efficiency by avoiding multiple cycles of operand collection, execution, and writeback on the SIMT lanes, avoiding multi-cycle functional unit occupancy in the SIMT lanes, and enabling the use the smaller and more efficient ASRF instead of the larger SRF. For microarchitectures which execute instructions over multiple cycles on the SIMT lanes, affine arithmetic will reduce execution to a single cycle.

Affine Branches are branches where the operands used in the branch comparison are affine. When the branch operand is uniform, a single comparison using the affine functional unit in the CP sufficiently applies to all threads in a warp. Uniform branches are common for inner loops within each thread. Comparisons between a uniform value and an affine value can be resolved on the CP if the uniform value is greater than or less than all elements in the affine sequence, implying a coherent branch resolution. Affine branches are common when performing bounds checking in SIMT kernels (e.g., stencil operations).

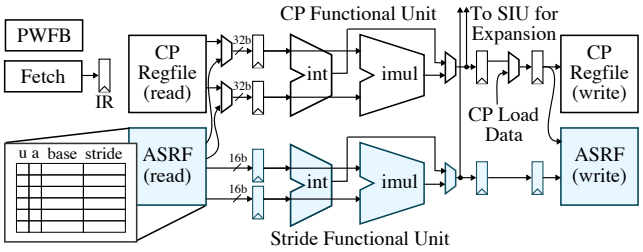

Figure 4: FG-SIMT CP with Compact Affine Execution – The ASRF and stride functional units are added to the baseline CP.

Uniform branches are implemented in both our GP-SIMT and FG-SIMT microarchitectures, but currently only GP-SIMT supports more general affine branches. In GP-SIMT, affine branches reduce pressure on the operand collectors. In FG-SIMT, uniform branches decrease the branch resolution latency by eliminating communication with the SIMT lanes. In both cases, energy efficiency is improved by avoiding reading the SRF and redundant comparisons across all threads in a warp.

Affine Memory Operations are load instructions with an affine base address. In FG-SIMT, the affine base address must also indicate a "unit-stride access" (i.e., $s = \{1,2,4\}$ for byte, halfword, and word accesses respectively). Affine memory operations are still sent to the SIU but are allowed to skip address generation on the SIMT lanes. In GP-SIMT, they can also bypass the operand collection stage since the source address is read and expanded from the ASRF. Affine memory operations are issued to the SMU immediately, since the access pattern is known from the affine base address. Like coalesced memory requests, affine memory operations use a wide request port; a full-warp affine memory operation is broken down into multiple cache-line width memory requests. The number of memory requests depends on the element width and cache alignment. For example, affine load word operations with 32 threads/warp and 32-byte cache lines require four wide accesses if the requests are aligned, and five wide accesses otherwise. A rotation network is used to assign the elements in each wide response to the appropriate lane. An extra memory command queue coordinates writeback between affine memory operations and other memory requests. Careful consideration is taken to avoid subtle memory ordering issues between the wide port and the narrow ports.

In GP-SIMT, a sophiscated, albeit expensive, coalescing unit provides full-warp coalescing, so the main benefit from affine memory operations is again from the reduced pressure on the operand collectors which affine memory operations can skip over. Note that affine memory operations can efficiently exploit value structure across a warp without the area and energy overhead of a complex coalescing unit. Affine memory requests are also injected into the memory system earlier. However, they are still issued to the SIMT pipeline rather than the affine pipeline to preserve memory ordering. We use a simpler coalescing unit in FG-SIMT which can only merge requests across lanes, and this allows affine memory operations to more significantly improve performance by more efficiently capturing full-warp vector memory operations. For byte- and half-word loads, this enables servicing the entire warp from potentially a single cache line access. Perhaps more importantly, affine memory operations avoid the energy required to redundantly compute 32 effective addresses and then dynamically detect that these are consecutive addresses. Both affine memory operations and coalesced requests eliminate cache bank conflicts by consolidating requests to consecutive addresses into a minimal number of accesses, improving both performance and energy efficiency.

4.3 Lazy Expansion Divergence Scheme

The basic scheme outlined in the previous section works only when the threads are converged. When threads diverge into multiple warp fragments, each fragment essentially needs its own set of affine registers; otherwise, one warp fragment could overwrite an entry in the ASRF with a new affine value before a different warp fragment has read the original affine value. Note that an important feature of our technique is that we can still efficiently execute affine branches and memory operations after divergence; the issue revolves around handling affine arithmetic after divergence. The key to exploiting affine arithmetic while threads are converged while still enabling correct execution after divergence is to carefully manage *affine expansion* after divergence. A naive scheme might use *eager affine expansion* to expand all values in the ASRF after the first branch diverges. Preliminary investigation of eager expansion determined that this caused unacceptable overhead; a significant amount of work was spent expanding values that were never used. In addition, eager expansion does not exploit the fact that we can still perform compact affine execution after divergence; we just cannot write to the ASRF after divergence.

Our single-ASRF divergence scheme uses *lazy affine expansion* of compact values. After divergence, affine arithmetic is still performed in the front-end, but the result is sent to the SIMT lanes to be expanded instead of being written back to the ASRF. The corresponding tag in the ASRF is updated to indicate that the destination registers is now generic. A subtle issue arises if the destination register in this situation was originally in the ASRF; we must expand the the original affine value for the non-active threads and then write the new value for the active threads to ensure that later warp fragments will read the proper value. Predicated instructions also require careful consideration; we must expand the destination even if they are not diverged since some threads might not execute based on data-dependent conditionals.

We have implemented the single-ASRF divergence scheme in both the GP-SIMT and FG-SIMT microarchitectures. This scheme can still achieve some of the benefits of compact affine execution after divergence, but also introduces overhead when we must lazily expand results. The actual performance of this scheme is heavily dependent on how early the workload diverges and how much opportunity for compact affine execution there is before divergence.

5. EVALUATION FRAMEWORK

This section describes the infrastructure used to evaluate compact affine execution in both GP-SIMT and FG-SIMT architectures.

5.1 GP-SIMT Hardware Modeling

We use a modified version of GPGPU-Sim 3.0 configured to model the GP-SIMT microarchitecture described in Section 3 with a PTX front-end and a realistic on-chip and off-chip memory system. GPGPU-Sim is a cycle-level microarchitectural model with a functional/timing split to enable rapid preliminary design space exploration [3]. We will evaluate two configurations: *gpsimt* without compact affine execution and *gpsimt+abm* which includes support for affine arithmetic, branches, and memory operations.

Our goal is to improve not only performance but also energy efficiency without negatively impacting area and cycle time. Unfortunately, it is difficult to accurately model execution time, energy, and area of real GPGPUs since the detailed microarchitecture of these systems is not public. We therefore limit ourselves to preliminary design space exploration using cycle-level modeling for GP-SIMT architectures and augment this with a much more detailed evaluation of FG-SIMT architectures.

5.2 FG-SIMT Hardware Modeling

We implemented a variety of different FG-SIMT configurations using RTL, and we use commercial ASIC CAD tools to enable: (1) cycle-accurate RTL simulation for design space exploration; (2) standard-cell synthesis and place-and-route for generating layout and estimating cycle time and area; and (3) gate-level simulation for estimating power consumption.

We use a combination of Synopsys DesignCompiler, IC Compiler, and PrimeTime PX along with a TSMC 40 nm standard cell library. We did not have access to a memory compiler for our target process, so we model tag/data SRAMs by creating abstracted "black-box" modules, with area, timing, and power models suitable for use by the CAD tools. We used CACTI [26] to explore a range of possible implementations and chose one that satisfied our design requirements. The rest of the cache is modeled using synthesizable RTL. Cache refill/eviction requests/responses are serviced by an idealistic functional main-memory model, and so we carefully configure our FG-SIMT benchmarks such that they fit in the L1 cache for the key timing loop.

For the evaluation, we compare the performance of the following designs at the RTL level: baseline FG-SIMT (*fgsimt*), FG-SIMT with a single-ASRF divergence scheme and affine arithmetic (*fgsimt+a*), affine arithmetic and branches (*fgsimt+ab*), and affine arithmetic, branches, and memory operations (*fgsimt+abm*). In addition, we also implemented an RTL model of an eight-core RISC processor (in-order, single-issue, one thread per core, no subword-SIMD units) with per-core private L1 instruction caches and a shared eight-bank L1 data cache (similar to the FG-SIMT design). The multicore configuration (*mcore*) is only provided as a useful reference point for the primary comparison between *fgsimt* and *fgsimt+abm*. Of these configurations, *mcore*, *fgsimt*, and *fgsimt+abm* were pushed through our ASIC toolflow for detailed evaluation of cycle time, area, and energy.

5.3 GP-SIMT and FG-SIMT Benchmarks

We have mapped a variety of benchmarks with diverse control, memory-access, and value structure to our GP-SIMT and FG-SIMT architectures (see Section 2.4, Tables 1–3). GP-SIMT benchmarks are drawn from the GPUGPU-Sim 3.0 distribution [3], Parboil suite [33], and Rodina suite [4] and also include custom benchmarks. FG-SIMT benchmarks mostly include custom benchmarks implemented using our lightweight C++ SIMT programming framework (see Section 2.1). In the rest of this section, we briefly describe these custom benchmarks.

bilat performs a bilateral image filter with a lookup table for the distance function and an optimized Taylor series expansion for calculating the intensity weight. *bsearch* uses a binary search algorithm to perform parallel look-ups into a sorted array of key-value pairs. *cmult* does floating-point complex multiplication across an array of complex numbers. *conv* is a 1D spatial convolution using a 20-element kernel. *dither* generates a black and white image from a gray-scale image using Floyd-Steinberg dithering. Work is parallelized across the diagonals of the image, so that each thread works on a subset of the diagonal. A data-dependent conditional allows threads to skip work if an input pixel is white. *kmeans* implements the k-means clustering algorithm [4] . Assignment of objects to clusters is parallelized across objects. The minimum distance between an object and each cluster is computed independently by each thread and an atomic memory operation updates a shared data structure. Cluster centers are recomputed in parallel using one thread per cluster. *mfilt* does a masked blurring filter across an image of

grayscale pixels. *rgb2cmyk* performs color space conversion on a test image. *rsort* performs an incremental radix sort on an array of integers. During each iteration, individual threads build local histograms of the data, and then a parallel reduction is performed to determine the mapping to a global destination array. Atomic memory operations are necessary to build the global histogram structure. *strsearch* implements the Knuth-Morris-Pratt algorithm to search a collection of byte streams for the presence of substrings. The search is parallelized by having all threads search for the same substrings in different streams. The deterministic finite automatas used to model substring-matching state machines are also generated in parallel. *viterbi* decodes frames of convolutionally encoded data using the Viterbi algorithm. Iterative calculation of survivor paths and their accumulated error are parallelized across paths. Each thread performs an add-compare-select butterfly operation to compute the error for two paths simultaneously, which requires unpredictable accesses to a lookup table.

Name	Description	GP	FG
aes	Advanced encryption standard	G	
bfs	Breadth-first search	R	P
bilat	Bilateral image filtering	C	C
bsearch	Parallel binary searches in linear array	C	C
cmult	Vector-vector complex multiplication	C	C
conv	Spatial convolution with large kernel	C	C
cutcp	Distance-cutoff coulombic potential	P'	
dither	Floyd-Steinberg image dithering		C
kmeans	KMeans clustering	R	C
lps	Laplace discretisation w/ Jacobin iteration	G	
mfilt	Masked image blur filter	C	C
nnet	Artificial neural network	G	
nqueens	N-Queens solver	G	
rgb2cmyk	RGB-to-CMYK color conversion		C
rsort	Radix sort of array of integers		C
sgemm	Dense matrix-matrix multiply	P	P
strsearch	Knuth-Morris-Pratt string search		C
viterbi	Viterbi decoder		C

Table 2: SIMT Benchmarks – GP = benchmarks for GP-SIMT; FG = benchmarks for FG-SIMT. C = custom implementation; G = implementation included in GPGPU-Sim distribution [3]; P = implementation adapated from Parboil suite [33]; P' = Parboil implementation included in GPGPU-Sim distribution; R = implementation from Rodinia suite [4].

6. EVALUATION RESULTS

In this section, we evaluate our approach using cycle-level GP-SIMT modeling and RTL/VLSI FG-SIMT modeling to make a case for exploiting value structure.

6.1 GP-SIMT: Cycle-level Performance

Figures 5 and 6 shows the performance of GP-SIMT with affine arithmetic, branches, and memory operations (*gpsimt+abm*) compared to the baseline (*gpsimt*) with a realistic memory system and warmed-up caches, respectively. A DRAM latency of 120 cycles was used for Figure 5. Results for a 16-lane (*L16*) and 8-lane (*L8*) GP-SIMT configuration with a slow and fast front-end are shown in both figures. We also studied a 32-lane design, where each warp occupies a functional unit for a single cycle. Affine execution yielded less benefit in this context, since warp instructions take the same number of cycles to execute in both the affine and SIMT pipelines. However, the primary performance benefit of affine execution is derived from the ability to skip the operand collection and writeback stages of the SIMT pipeline, as well as the reduced occupancy of the SIMT functional units. Note that reducing the number of lanes might not always translate to better performance using affine execution because multithreading often adequately hides functional unit latencies. Affine execution also helps to keep the functional units busy more often as seen by the increasing average warp issue rate for all applications.

We expect to see higher speedups in *L8* which executes instructions over more cycles; affine execution reduces this to a single cycle. However, Figure 5 shows that this is not always the case. This can be attributed to the changing memory access patterns related to the varying microarchitectural latencies which can significantly affect performance. For instance, applications like *cmult* or *conv* which are dominated by instructions dependent on multiple critical loads are especially sensitive to issue scheduling. Factoring out this sensitivity to the memory system yields results more in line with expectations as shown in Figure 6. Compute-limited applications with close dependencies between instructions, such as *conv* and *sgemm*, see more benefit from affine execution on *L8* compared to *L16*. However, in many cases, warp multithreading is effective in eliminating such dependencies, dampening the benefits of affine execution.

The results for the slower front-end are shown to motivate the need for a faster front-end for affine execution. With a slower front-end, the affine functional units take the same number of cycles to execute a warp instruction as the SIMT lanes. This severely limits the ability of affine execution to more quickly reach the critical non-affine instructions. It also hides the benefits of reducing the number of cycles for a warp instruction to execute (i.e., affine execution on *L8* will not show more benefit than *L16*). Generally, a faster front-end will yield higher speedups for compute-heavy applications with a high density of affine instructions. Note that a faster front-end can also benefit the baseline GP-SIMT which can affect the relative speedup as well.

Table 1 lists statistics about the GP-SIMT results which will be used in the following analysis. In general, applications with more opportunities for affine execution show greater speedups. About half the applications show notable speedups ranging from roughly 8–30%, a quarter with marginal or no speedup, and the rest showing slightly worse performance. With a realistic memory system as shown in Fig 5, cache misses and high memory latencies can dominate the performance of many applications, overshadowing the benefits of affine execution. We offer a set of results with warmed-up caches to isolate the effects of affine execution.

nnet has the highest density of affine instructions of all the applications. Much of the computation is on a per-block basis, making it ideal for affine execution. *nnet* also is dominated by address computation and complex mathematical approximations (i.e., tanh) which introduce many affine values. As an aside, note that *nnet* has just 1.9 active threads on average. This is due to very short application vector lengths as opposed to divergence.

Applications such as *bfs*, *kmeans*, *lps*, and *median* suffer from high degrees of thread divergence which suppress affine execution and add the overhead of affine expansions. *kmeans* in particular has the highest expansion overhead at 15%, negating the effects of affine execution and degrading performance. *cutcp* is an interesting case as it has a high percentage of affine instructions but shows no improvement. In this case, performance is dominated by non-affine long latency operations, with the highest percentage of all applications at 35%. Similarly, in *nqueens*, the serialized reduction stages dominate and overshadow the benefits of affine execution. It is worth noting that applications with lower average warp

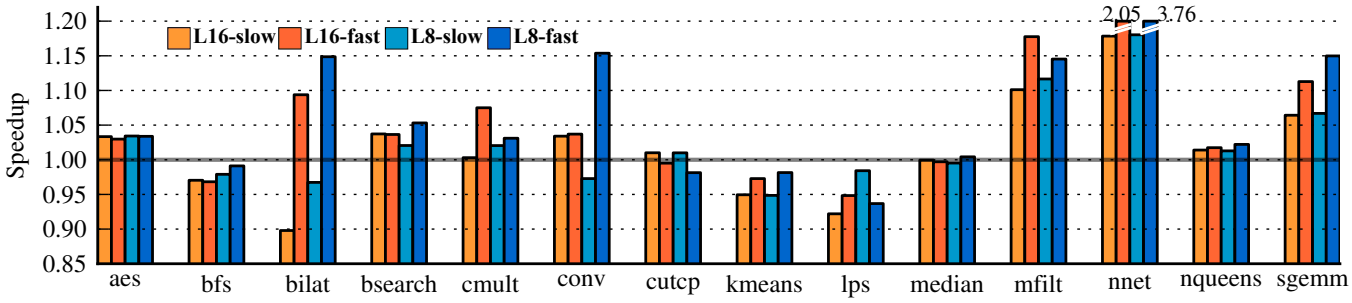

Figure 5: GP-SIMT Cycle-Level Performance with Realistic Memory – Each bar shows the speedup for GP-SIMT with all affine extensions (*gpsimt+abm*) relative to a baseline (*gpsimt*) with the same number of lanes (L8 = 8 lanes, L16 = 16 lanes) and front-end configuration.

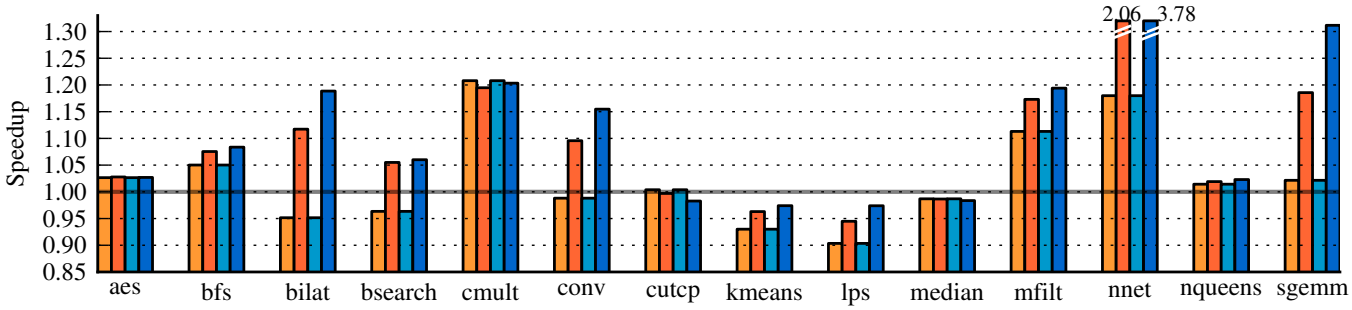

Figure 6: GP-SIMT Cycle-Level Performance with Warmed-Up Caches – Each bar shows the speedup for GP-SIMT with all affine extensions (*gpsimt+abm*) relative to a baseline (*gpsimt*) with the same number of lanes (L8 = 8 lanes, L16 = 16 lanes) and front-end configuration. All datasets are sized to fit in the cache.

	gpsimt							gpsimt+abm							
Name	Dyn inst	Act thds	Avg iss	LL inst	Ctrl inst	Mem inst	Div inst	Warp issue	Aff inst	Supp aff	Exp inst	Aff rds	Aff wrs	Aff bra	Aff mem
aes	30.1	32.0	0.5	11%	0%	25%	0%	0.5	29%	0%	0%	21%	28%	0%	53%
bfs	14.0	10.3	0.5	7%	8%	32%	78%	0.6	17%	52%	9%	24%	38%	91%	62%
bilat	7.2	31.1	0.9	24%	7%	7%	24%	1.1	39%	18%	6%	24%	40%	100%	18%
bsearch	0.2	25.0	0.6	4%	16%	12%	77%	0.6	9%	66%	5%	9%	14%	98%	78%
cmult	0.6	32.0	0.6	19%	3%	45%	~0%	0.8	52%	0%	0%	38%	52%	0%	14%
conv	2.1	31.9	1.1	10%	10%	28%	~0%	1.2	46%	~0%	~0%	32%	50%	91%	32%
cutcp	126.0	32.0	0.6	35%	5%	25%	~0%	0.6	45%	~0%	~0%	26%	45%	0%	~0%
kmeans	2.6	25.1	0.4	5%	5%	10%	66%	0.5	24%	52%	15%	16%	22%	85%	3%
lps	72.7	24.2	1.1	10%	12%	23%	92%	1.2	5%	86%	8%	~0%	5%	~100%	75%
median	0.3	20.1	0.8	1%	13%	3%	80%	0.8	5%	78%	3%	6%	9%	100%	34%
mfilt	2.5	29.5	0.4	13%	7%	24%	46%	0.5	41%	10%	3%	31%	41%	55%	38%
nnet	22.4	1.9	0.8	17%	8%	1%	~0%	1.7	98%	~0%	~0%	70%	77%	~0%	1%
nqueens	1.2	25.8	0.4	3%	11%	29%	29%	0.5	42%	16%	2%	55%	70%	42%	28%
sgemm	9.3	32.0	0.8	22%	1%	23%	0%	0.9	13%	0%	0%	9%	12%	0%	2%

Table 1: Statistics for GP-SIMT Benchmarks Dyn inst = dynamic instructions in millions; Act thds = avg. active threads per warp (max=32); Avg iss = avg. instructions issued per cycle (max=2); LL, Ctrl, Mem inst = % long latency, branch/jump, load/store instructions; Div inst = % instructions under divergence; Aff inst = % affine instructions (not including suppressed); Supp aff = % affine instructions suppressed by divergence; Exp inst = % overhead of affine expansion; Aff rds/wrs = % reduction of SRF reads/writes; Aff bra, mem = % affine branches, loads; Values rounded to 0% or 100% are prefixed with '~'.

instruction issue rates are usually bottlenecked by non-affine long-latency operations or the memory system, which overshadows the benefits gained from affine execution. The percent of affine instructions suppressed by divergence range from 10–86% for divergent benchmarks, suggesting future optimizations for exploiting value structures after divergence could further improve performance.

As a proxy for the energy savings provided by affine execution, Table 1 also shows the percent of SRF reads/writes saved by accessing the more compact ASRF. In most cases, affine execution replaces 16–55% of SRF reads and 11–52% of SRF writes with more efficient accesses to the ASRF. Note that these savings also apply to the 32-lane configuration.

Overall, these results provide additional motivation for our detailed FG-SIMT evaluation to better quantify the cycle time, area, and energy overheads associated with our approach.

6.2 FG-SIMT: Cycle Time, Area Comparison

Figure 7 shows the area breakdown of the placed-and-routed designs. Amortizing the instruction fetch over a warp reduces the instruction cache size in *fgsimt* compared to *mcore*. However, *fgsimt* pays a significant overhead for the large SRF. The additional wide port for memory coalescing and affine memory operations adds area overhead to the crossbar. The ASRF, stride functional unit, and expansion logic of *fgsimt+abm* slightly increases the area of the CP and SIMT lanes.

The critical path of *mcore* is through the memory system. The critical path of both the FG-SIMT designs are through the SRF read and address generation. The marginal difference in cycle time of *fgsimt+abm* and *fgsimt* is within the variability margin of the ASIC CAD tools.

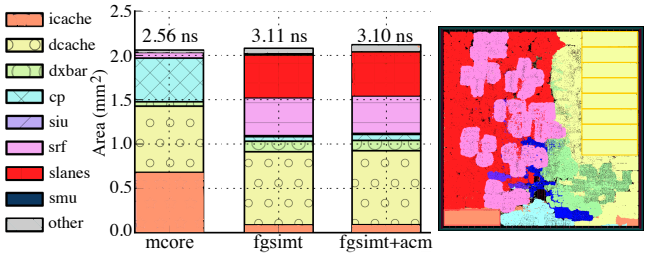

Figure 7: Area Breakdown – (left) absolute area breakdowns of *fgsimt+abm*, *mcore*, and *fgsimt*. Bars are annotated with cycle times; (right) layout of placed-and-routed *fgsimt+abm* implementation in a TSMC 40nm process.

6.3 FG-SIMT: RTL Results

Figure 9 shows the RTL cycle count speedups of *fgsimt+abm* compared to the baseline *fgsimt*. The trends are similar to those found in the GP-SIMT results with some noticeable differences. Table 3 shows the various metrics used to quantify the benefits gained from each proposed mechanism.

For affine arithmetic, the percent of affine instructions reflects the amount of work that is amortized across an entire warp by executing an instruction affinely on the CP. In addition, we obtain higher energy efficiency by reading once from the ASRF instead of having every active thread read from the SRF, as shown by the increased percentage of affine reads and writes. Affine branches avoid the expensive branch resolution latency of the SIMT lanes which FG-SIMT cannot hide with multithreading, attributing to the more pronounced performance increase of *fgsimt+ab*. Affine memops reduce the total number of memory requests and reduce bank conflicts; this is reflected in the table as the percentage of memops which we can infer to be vector memops. The results suggest that affine memops can further improve the efficiency gained from coalescing. Overall, exploiting value structure using compact affine execution in *fgsimt+abm* improves performance over *fgsimt* for a majority of our benchmarks, with speedups ranging from 20–65% for those with significant improvement. The degree of improvement with each added mechanism depends on the amount of value structure present in the corresponding class of instruction as characterized in Section 2.

One interesting difference from the GP-SIMT results is *kmeans* for which FG-SIMT shows a pronounced improvement. This is partly because the FG-SIMT optimized *kmeans* eliminates some of the divergence we saw in GP-SIMT. In *bilateral*, the lack of multithreading in FG-SIMT makes it difficult to hide the latency of long-latency floating point operations which throttle the performance. However, note that in the general case, FG-SIMT is still able to obtain significant speedups without multithreading, which also means it does not incur the associated, non-trivial overheads. Each mechanism only improves or does not affect performance. The rare exception to this is inferred vmemops, as shown by *rsort*, which occurs when the altered memory access pattern ends up causing more bank conflicts between narrow and wide requests.

Minimal or no improvement is seen for the most divergent benchmarks where execution is dominated by branch resolution (e.g. average number of active threads 13.2 and 17.1, for *strsearch* and *bfs*, respectively). Although *kmeans* is quite divergent, the higher density of uniform branches mitigate the performance loss. *rsort* is not divergent, but uses small number of threads so that *fgsimt* cannot take advantage of the shared load cache. The shared load cache is ineffective in this case since the cache is invalidated after the end of each kernel to prevent coherence issues.

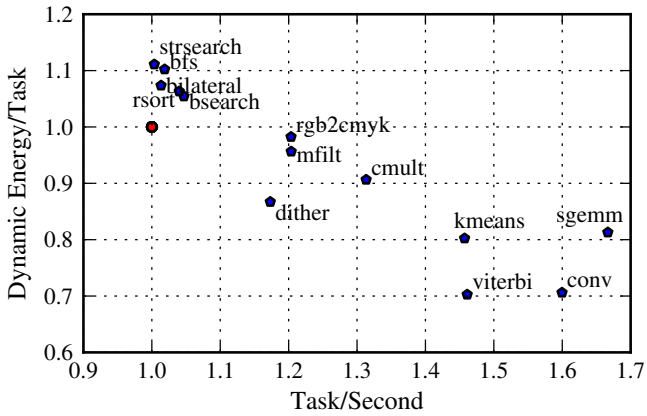

Figure 8: FG-SIMT Normalized Dynamic Energy vs. Performance – Results are for *fgsimt+abm* normalized to *fgsimt* annotated on (1,1).

6.4 FG-SIMT: Energy vs. Performance

Figure 8 shows the dynamic energy and performance of *fgsimt+abm* relative to *fgsimt*. A majority of the benchmarks show energy reductions, with maximum savings up to nearly 30%. Overall, compact affine execution reduces the amount of work to complete the same task. For example, in one of the best cases, *viterbi* shows a 54% decrease in the combined register file (SRF + ASRF) energy, a 29% decrease in the combined functional unit (SIMT lane units + CP affine units) energy, and a 34% reduction in the memory system energy by enabling affine execution.

Divergent benchmarks can incur a significant dynamic energy overhead from affine expansions which are triggered when affine registers are overwritten after divergence. The worst case, *strsearch*, exemplifies this overhead. The energy consumption in *strsearch* increases from 115uJ to 130uJ, a percent increase of 12%. Of the 15uJ increase, 89% is attributed to the SIMT lanes, specifically the VAUs which manages affine expansion. In such cases, compact affine execution is not amortized across enough threads, so that the energy overheads of affine expansion outweigh the marginal performance benefit.

7. RELATED WORK

There have been numerous proposals for microarchitectural mechanisms to improve the performance of SIMT architectures including: dynamically creating new warp fragments to better handle irregular control flow and memory accesses [24, 34]; dynamically merging warp fragments into new warps to mitigate control divergence [12, 13, 30]; new reconvergence schemes [6, 10]; and new SIMT register file organizations [14]. These techniques are all complementary and orthogonal to the mechanisms presented in this paper.

The closest work to our own is a preliminary study by Collange et al. in [7] which uses a cycle-level simulator to track the number of affine values read and written to the SIMT register file for CUDA benchmarks. Similar to our own study in Section 2 for FG-SIMT benchmarks, Collange at al. found significant value structure that is currently not exploited in SIMT architectures. This prior work also describes a scheme for tracking affine values derived from shared loads and thread indices similar to our own (although we also handle affine multiplications) and sketches a possible way to exploit this value structure using the SIMT register file to hold affine values and the SIMT functional units to perform affine arithmetic. We

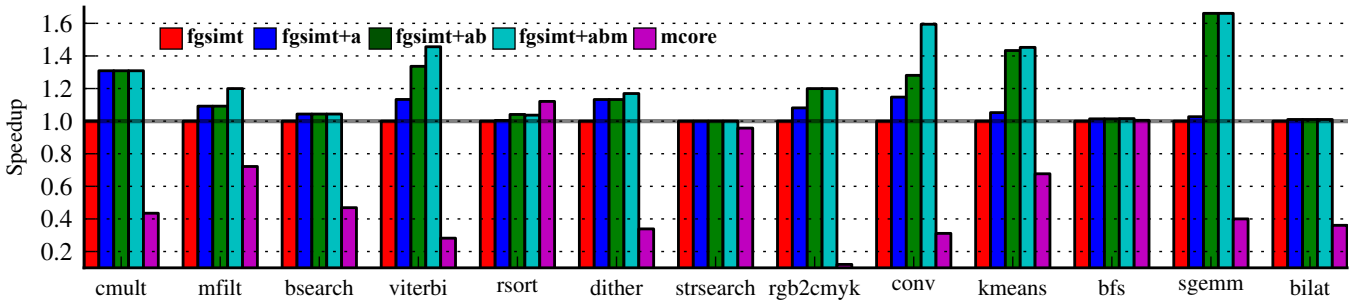

Figure 9: FG-SIMT Cycle-Level Performance – Results for all affine configurations and *mcore* normalized to *fgsimt*.

	mcore			fgsimt									fgsimt+a					fgsimt+ab		fgsimt+abm			
Name	Dyn inst	Exec time	Power (mW)	Dyn inst	Int inst	FP inst	Ctrl inst	Mem inst	Act thds	Mem B/cyc	Exec time	Power (mW)	SL inst	Aff inst	Aff rds	Aff wrs	Exec time	Aff inst	Exec time	Aff inst	Aff mem	Exec time	Power (mW)
bfs	16	46	120	53	48%	0%	16%	32%	17.1	1.9	46	94	1%	4%	34%	12%	46	4%	46	4%	1%	45	106
bilateral	860	3058	108	6304	29%	48%	6%	15%	31.3	3.7	1103	125	4%	8%	22%	5%	1092	8%	1092	8%	~0%	1092	136
bsearch	28	47	145	207	78%	0%	4%	14%	28.0	5.9	22	135	2%	10%	27%	10%	21	10%	21	10%	~0%	21	149
cmult	2	7	133	20	20%	27%	1%	42%	31.3	12.5	2.9	168	13%	55%	64%	38%	2.2	55%	2.2	55%	6%	2.2	200
conv	25	88	115	171	34%	19%	11%	32%	31.0	9.2	27	139	3%	53%	83%	54%	24	63%	21	63%	3%	17	157
dither	1054	2018	129	3414	62%	0%	3%	30%	28.9	6.1	684	116	12%	44%	64%	49%	603	44%	603	44%	5%	584	118
kmeans	74	221	123	522	40%	15%	13%	29%	14.0	4.1	149	130	1%	51%	49%	33%	142	62%	104	62%	8%	103	152
mfilt	26	74	119	274	52%	0%	3%	38%	24.9	8.2	54	119	13%	44%	68%	43%	49	44%	49	44%	3%	45	137
rgb2cmyk	57	167	122	185	66%	0%	4%	29%	26.7	11.1	20	181	~0%	25%	54%	17%	18	29%	16	29%	3%	16	214
rsort	43	101	129	290	52%	0%	14%	32%	30.9	3.5	113	104	~0%	5%	38%	9%	112	6%	108	6%	1%	109	115
sgemm	34	97	130	252	39%	22%	11%	25%	32.0	7.5	38	124	2%	14%	44%	16%	37	26%	23	26%	0%	23	168
strsearch	200	379	136	1237	70%	0%	13%	15%	13.2	3.1	363	113	0%	6%	~0%	~0%	363	6%	363	6%	1%	363	126
viterbi	947	2156	121	3819	61%	0%	6%	29%	32.0	7.5	608	113	8%	42%	78%	65%	536	45%	455	45%	3%	417	116

Table 3: Statistics for FG-SIMT Benchmarks – Dyn inst = dynamic instructions in thousands; Exec time = execution time in thousands of cycles; Power = avg. power in mW; Int, FP, Ctrl, Mem, SL inst = % integer, floating point, branch/jump, load/store, shared load instructions; Act thds = See Table 1; Mem B/cyc = avg. L1 memory traffic in bytes/cycle (max=32B/cyc); Aff inst, rds/wrs, mem = See Table 1; Values rounded to 0% or 100% are prefixed with '~'.

found this approach to be difficult to implement in practice, and it does not enable compact affine execution to take advantage of a more energy efficient register file dedicated for holding affine values. This prior work does not handle affine multiplication, support non-power of two strides, describe a way to manage the interaction between affine execution and divergence, nor quantitatively investigate the cycle time, area, and energy impact of the proposed techniques. In addition, our own approach supports efficient execution of affine branches with both uniform and affine operands as well as affine memory operations enabling dynamically generating vector memory operations without the need for memory coalescing. We see the prior work in [7] as useful characterization of the potential for compact affine execution, and valuable motivation for the mechanisms proposed in this paper.

Other prior work on exploiting value structure is either more limited or orthogonal to our own. A scheme briefly mentioned by Gilani et al. uses many cross-lane comparators to dynamically detect uniform values across all threads in a warp [15]. Our own approach achieves the same effect in the common case by more efficiently tracking value structure derived from shared loads and the thread index. It is true, however, that actually comparing all values across all threads in a warp is more general since this can detect value structure in load/store data or in register values after divergence (although we did not find these situations to be very common). The affine vector cache proposed by Collange et al. exploits value structure in load/store data, and specifically increases the prevalence of this structure by overconstraining register usage and forcing register spilling [8]. Since compact affine values can be stored as base/stride pairs in the memory system, this can enable more warps in-flight by reducing the number of registers required by each thread. This technique is orthogonal to our own, and an interesting direction for future work would be to investigate the interaction between these two proposals.

There has additionally been a considerable amount of compiler work for optimizing SIMT code, including optimizations which leverage value structure detection to enable redundant code removal [5, 9, 11, 19, 21, 31, 32]. Collange proposed a compiler analysis pass for detecting value, memory, and control structure, including a technique called scalarization to detect scalar instructions [5]. Other work proposed divergence analysis which statically detects uniform and affine variables in CUDA code [9, 31]. Compiler techniques can be effective in simple cases where a single value is shared across all threads, but it is challenging to statically exploit more sophisticated value structure especially after control flow [21]. The microarchitectural mechanisms presented in this paper can complement SIMT compiler techniques; compiler optimizations can potentially simplify our implementation, while compilers can now take advantage of the more sophisticated compact affine execution resources available in our design.

There has also been work on exploiting value structure in non-SIMT architectures. Long et al. introduced minimal multithreading, a microarchitectural mechanism to eliminate redundant computation in SPMD applications [23]. Unlike our technique which assumes a data-parallel SIMT architecture, minimal multithreading builds on top of a multi-threaded SMT core. Lee et al. proposed vector-threading as an alternative approach to SIMT-based data-parallel accelerators [20]. Vector-threading enables manually exploiting value structure by allowing the programmer to factor out redundant computation onto the architecturally exposed control processor. In traditional subword/vector-SIMD architectures,

value structure is also often exploited in the compiler by refactoring work onto a scalar control processor and mixing scalar and vector computation at a fine granularity. SIMT microarchitectures usually lack a software-exposed control processor or a fine-grain mechanism for mixing scalar and vector computation, thus threads within a warp often spend time and energy inefficiently executing operations on well-structured values. While scalar work can be refactored onto FG-SIMT's software-exposed control-processor, the techniques proposed in this paper simplify the compiler and allow more complicated code to be easily grouped into a single FG-SIMT kernel; the hardware will dynamically handle executing what is traditionally refactored onto the CP and exploit dynamic information for further improvements.

8. CONCLUSIONS

In this paper, we proposed mechanisms to exploit value structure in SIMT architectures that enable efficiently tracking and executing affine arithmetic, branches, and memory operations. The proposed mechanisms can be applied to the GP-SIMT microarchitectures found in graphics processing units as well as FG-SIMT microarchitectures, a SIMT variant targeted towards compute-focused data-parallel accelerators. We discussed many of the implementation details required to make this approach feasible (e.g., lazy expansion of compact values, handling affine branch operands). Interesting potentials for future work include exploiting value structure more effectively after divergence, compactly encoding value structure in branch resolution, optimizing affine execution to better tolerate significant memory latencies, and adaptive scheduling based on the presence of affine values. This work demonstrated the promise of this approach in improving both the performance and energy-efficiency of SIMT architectures within the context of a more traditional GP-SIMT cycle-level model and a more detailed FG-SIMT VLSI implementation.

ACKNOWLEDGMENTS

This work was supported in part by NSF CAREER Award #1149464, a DARPA Young Faculty Award, an NDSEG Fellowship, and donations from Intel Corporation, NVIDIA Corporation, and Synopsys, Inc. The authors acknowledge and thank Christopher Fairfax and Berkin Ilbeyi for their help in writing application kernels.

REFERENCES

[1] HD 6900 Series Instruction Set Architecture, Rev 1.1. AMD Reference Guide, Nov 2011.

[2] AMD Graphics Cores Next Architecture. AMD White Paper, 2012.

[3] A. Bakhoda et al. Analyzing CUDA Workloads Using a Detailed GPU Simulator. *ISPASS*, Apr 2009.

[4] S. Che et al. Rodinia: A Benchmark Suite for Heterogeneous Computing. *IISWC*, Oct 2009.

[5] S. Collange. Identifying Scalar Behavior in CUDA Kernels. Technical Report HAL-00622654, ARENAIRE, Jan 2011.

[6] S. Collange. Stack-less SIMT Reconvergence at Low Cost. Technical Report HAL-00622654, ARENAIRE, Sep 2011.

[7] S. Collange et al. Dynamic Detection of Uniform and Affine Vectors in GPGPU Computations. *Workshop on Highly Parallel Processing on a Chip*, Aug 2009.

[8] S. Collange et al. Affine Vector Cache for Memory Bandwidth Savings. Technical Report ENSL-00622654, ENSL, Dec 2011.

[9] B. Coutinho et al. Divergence Analysis and Optimizations. *PACT*, Oct 2011.

[10] G. Diamos et al. SIMD Re-Convergence at Thread Frontiers. *MICRO*, Dec 2011.

[11] G. Diamos et al. Ocelot: A Dynamic Compiler for Bulk-Synchronous Applications in Heterogenous Systems. *PACT*, Sep 2010.

[12] W. W. Fung et al. Thread Block Compaction for Efficient SIMT Control Flow. *HPCA*, Feb 2011.

[13] W. W. Fung et al. Dynamic Warp Formation: Efficient MIMD Control Flow on SIMD Graphics Hardware. *ACM Trans. on Architecture and Code Optimization*, 6(2):1–35, Jun 2009.

[14] M. Gebhart et al. Energy-Efficient Mechanisms for Managing Thread Context in Throughput Processors. *ISCA*, Jun 2011.

[15] S. Z. Gilani et al. Power-Efficient Computing for Compute-Intensive GPGPU Applications. *HPCA*, Feb 2013.

[16] Intel OpenSource HD Graphics Programmer's Reference Manual, Vol 4, Part 2, Rev 1.0. Intel Reference Manual, May 2011.

[17] Introducing Intel Many Integrated Core Architecture. Intel Press Release, 2011.

[18] Intel SDK for OpenCL Applications: Optimization Guide. Intel Reference Manual, 2012.

[19] A. Kerr et al. Dynamic Compilation of Data-Parallel Kernels for Vector Processors. *CGO*, Apr 2012.

[20] Y. Lee et al. Exploring the Tradeoffs between Programmability and Efficiency in Data-Parallel Accelerator Cores. *ISCA*, Jun 2011.

[21] Y. Lee et al. Convergence and Scalarization for Data-Parallel Architectures. *CGO*, Feb 2013.

[22] E. Lindholm et al. NVIDIA Tesla: A Unified Graphics and Computer Architecture. *IEEE Micro*, 28(2):39–55, Mar/Apr 2008.

[23] G. Long et al. Minimal Multi-Threading: Finding and Removing Redundant Instructions in Multi-Threaded Processors. *MICRO*, Dec 2010.

[24] J. Meng et al. Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance. *ISCA*, Jun 2010.

[25] Graphics Guide for Windows 7: A Guide for Hardware and System Manufacturers. Microsoft White Paper, 2009.

[26] N. Muralimanohar et al. CACTI 6.0: A Tool to Model Large Caches, 2009.

[27] J. Nickolls et al. Scalable Parallel Programming with CUDA. *ACM Queue*, 6(2):40–53, Mar/Apr 2008.

[28] NVIDIA's Next Gen CUDA Compute Architecture: Fermi. NVIDIA White Paper, 2009.

[29] OpenCL Specification, v1.2. Khronos Working Group, 2011.

[30] M. Rhu et al. CAPRI: Prediction of Compaction-Adequacy for Handling Control-Divergence in GPGPU Architectures. *ISCA*, Jun 2012.

[31] D. Sampaio et al. Divergence Analysis with Affine Constraints. Technical Report HAL-00650235, ARENAIRE, Dec 2011.

[32] J. A. Stratton et al. Efficient Compilation of Fine-Grained SPMD-Threaded Programs for Multicore CPUs. *CGO*, Apr 2010.

[33] J. A. Stratton et al. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. Technical report, UIUC, IMPACT-12-01, Mar 2012.

[34] D. Tarjan et al. Increasing memory miss tolerance for SIMD cores. *Supercomputing*, Aug 2009.

[35] M. Yuffe et al. Fully Integrated Multi-CPU, GPU, and Memory Controller 32 nm Processor. *ISSCC*, Feb 2011.